OR-PARALLEL PROLOG ON SHARED MEMORY MULTIPROCESSORS

ANDRZEJ CIEPIELEWSKI, SEIF HARIDI, AND BOGUMIL HAUSMAN

▷ PROLOG implementation efforts have recently begun to shift from single-processor systems to the new commercially available shared-memory multiprocessors. Among the problems encountered are efficient implementation of operations on variables and the scheduling of the processors. Most of the solutions proposed so far suffer from expensive, nonconstant-time implementation of operations on variables. We propose a storage model (versions-vector model) and a scheduling algorithm. The objectives of the scheduling algorithm are to approximate the sequential processing whenever feasible and to minimize the change in the state of a processor looking for a new task. The most important property of the storage model is a constant-time implementation of operations on all variables. The price paid for efficiency in managing variables is a nonconstant time of task switching. We propose three ways to decrease this price. The first is promotion of variables from versions vectors to value cells on the stack or heap during a task switch, making the subsequent task switches cheaper. The second is delayed installation of variables in versions vectors, decreasing the cost of short branches. The third is a possibility of restricting parallelism to predicates which can gain from the OR-parallel execution. ◁

1. INTRODUCTION

PROLOG implementation efforts have recently begun to shift from single-processor systems to the new commercially available shared-memory multiprocessors. A typical shared-memory multiprocessor has up to 30 processors. The challenge is to utilize this class of multiprocessor systems in such a way that most programs will run much faster and no program will run much slower than on single-processor systems [20].

Address correspondence to Bogumil Hausman, Swedish Institute of Computer Science, Box 1263, S-164 28 Kista, Sweden.

Received June 1987; accepted February 1988

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/89/$3.50

There are two main sources of parallelism in PROLOG programs: one is nondeterminism in the choice of goals, and the other, nondeterminism in the selection of clauses. The former is the source of AND parallelism, and the latter of OR parallelism [7]. The two sources are difficult to combine. One drastic solution is to give up the possibility of producing several solutions and thus restrict OR parallelism, as is done in the committed-choice languages [6,13,15]. In this way some power of PROLOG is lost. Another way is to keep the power of PROLOG, but partly sequentialize either the processing of goals, or that of clauses, or both. We have chosen to construct an OR-parallel system which handles don't-know nondeterminism by a combination of the breadth-first (parallel) and depth-first (sequential) strategies. The system not only produces multiple solutions in parallel, it also supports reasonably efficient AND-parallel execution of independent goals with small solution sets. The AND parallelism is obtained by transforming a parallel conjunction to a combination of disjunctions.

There are several good reasons for beginning with an OR-parallel system. OR parallelism seems to offer good potential for large-scale, large-granularity parallelism across a wide range of applications such as querying a deductive database, parsing a natural-language sentence, and compiling a set of objects [5]. Implementation of OR-parallel systems is closest to standard sequential implementation, and as such can utilize shared memory most efficiently, and also incorporate large parts of sequential systems, saving a lot of work and time. One of the goals of this paper is to show that OR parallelism can be implemented efficiently. Indeed, the experience from the prototype implementation [9] supports this conjecture. The parallel system runs on one processor only 20% slower than the state-of-the-art sequential system [2] from which it originated, and it shows close to linear speedups for programs with sufficient parallelism.

The specific problem with OR-parallel implementations is the management of simultaneous multiple bindings of variables. The number of different solutions (storage models) to the problem has proliferated during the last few years, starting from the abstract models by Pollard [12] and Ciepielewski and Haridi [3], through more and more implementation oriented models: Borgwardt [1], D. S. Warren [16], Lindstrom [8], and recently Tinker and Lindstrom [14] and Disz et al. [5].

In a recent paper [17] D.H. D. Warren organizes the field and reconstructs some of the models, deriving them from the classical "abstract model" of resolution theory. Warren proposes two important measures for comparing the storage models: the cost of creating and accessing variable bindings, and the cost of creating multiple tasks. In standard PROLOG implementations the creation of and access to bindings are very fast *constant-time* operations for all variables. The backtracking operation, which corresponds to task switching, takes typically 15–20 machine instructions, plus the time for "untrailing" variable bindings. In the models quoted above, the creation and access of bindings are *not* constant-time operations for some variables. On the other hand, the time for task creation can be made constant in most of the above models, except for untrailing when applicable. Warren proposes another model (SRI model) [17,18] in which creation of and access to bindings *are* constant-time operations for all variables, but task switching is not. The idea is to extend the conventional WAM [19] with a large binding array per processor and modify the trail to contain address-value pairs instead of just addresses. Each table is used by just *one* processor to store and access conditional

bindings. On a task switch, the table for the processor starting a new task must be partially reconstructed using the trail in the processor from which the task is taken ("stolen").

The solution we present in this paper has time characteristics similar to that of the SRI model, but instead of having one large table per processor, we propose using a vector of instances (versions vector) per shared variable. Each vector has the number of components equal to the number of processors in the system. Each component in a vector is used by just *one* processor to store and access one version of the variable's value. Like Warren, we assume that there are no more active branches than there are processors. We will show in Section 5 that the main difference between the SRI model and the VV model is the time at which space for processor specific bindings is allocated.

Our idea is related to Pollard's presented in [12] and [4]. The difference is that Pollard proposes using a tree of bindings per variable together with a fairly complex naming scheme for identifying the branch (not processor!) owning a binding.

One aim of this paper is to present new algorithms and implementation techniques leading to efficient implementations of OR parallelism. Another aim is to create a base for comparing different proposals, by specifying a set of basic operations on the level still well above the concrete implementation, but already allowing for meaningful estimation of the complexity of the different solutions.

The rest of this paper is organized as follows. In Section 2 we briefly describe a variant of the standard WAM in order to create a frame of reference and be able to show the relative cost of the following modifications. In Section 3 we give the scheduling algorithm which replaces backtracking. The algorithm is a mixture of the breadth-first and the depth-first strategies. In Section 4 we present our storage model and compare it with the SRI model. In Section 5 we specify the extended machine. In Section 6 we present the optimizations reducing the cost of task switching. In Section 7 we describe some restrictions on parallelism and their consequences. In Section 8 we discuss low-level aspects like locking and storage allocation.

2. SIMPLE WAM

One of our goals is to show that an OR-parallel implementation differs in relatively minor ways from a sequential one. The specification in this section is meant as a frame of reference and a remainder of the simplicity of operations in sequential implementations. Below we present a simplified version of the WAM used by SICS. We shall call it S-WAM for conciseness. The description captures only the properties changed in the extended WAM (VV-WAM) presented in the following sections.

Data Areas

The memory of the S-WAM consists of four areas: environment stack, term heap, trail, and control stack. In this paper we will treat the environment stack and the term heap together and call it stack. The S-WAM differs from the original WAM in having an explicit control stack for storing choice points. There are other differences, but they are of no importance here.

Registers

Part of the current computation state is held in a number of registers. We list only the registers relevant for this paper:

`P`	Program pointer
`ST`	Top of stack pointer
`B`	Top of control stack pointer
`TR`	Top of trail pointer

Contents of a Choice Point

A choice point is used to store the part of the machine state to be used at backtracking. For simplicity we make explicit some information (open alternatives) not explicitly present in the choice points of sequential implementations. We only show a part of a choice point's contents:

`P'`	Alternative clause pointer
`ST'`	Alternative top of stack
`TR'`	Alternative top of trail
`OA'`	Number of alternative clauses left (open alternatives)

`ST'` divides the stack into the private part created after the latest choice point, and the shared part created before. A new variable cell is always created in the private part of the stack. If the variable becomes bound while still in the private section, the binding is called unconditional; otherwise it is called conditional. Only conditionally bound variables are saved on the trail ("trailed"). A choice point with some alternatives left to be explored (`OA'` > 0) is called *open*; otherwise it is called *closed*.

Data Objects

A PROLOG term is represented by a word containing a tag and a value. The tag distinguishes the type of a term. We assume for simplicity that there are just two types of terms: variable terms (with the tag `VAR`) and non-variable terms (with the tag `NON-VAR`). An unbound variable is represented by a variable term bound to itself.

Conventions

In the description of operations we adopt mostly the C-language-like conventions, with the following additions. Machine registers and components of the topmost choice point are used as global variables. Their names are written with capital letters. Local variables have names starting with a lowercase letter. We use some abbreviations: `tag(V)` denotes the tag part of the term `V`, `value(V)` denotes the value part of the term `V`, `tagged(T,V)` denotes a tagged word (term) with the tag `T` and the value `V`. Finally `<<` and `>>` are operators comparing pointers. For example: if `U<<V` is true, then `U` points to an object on the stack (trail, control stack) that has been created earlier (is older) than the object pointed to by `V`.

Operations

The relevant operations are: **deref**—dereference a variable; **bind**—bind a variable to a value and possibly trail the address, **create_choice_point** (corresponds to TRY instruction in WAM)—create a new choice-point and save the current state; **backtrack** (corresponds to RETRY and TRUST)—untrail variables, choose the next alternative in the search tree, and possibly remove the current choice point. The operations are divided into operations on a single variable (**deref** and **bind**), and scheduling operations (**create_choice_point** and **backtrack**). Again we introduce a new name, scheduling, to put the reader in the right state of mind.

Operations on a Single Variable

```
deref(V) term *V
    {if tag(*V) == VAR and not value(*V) == V      % a bound variable
        then deref(*V)                             % follow chain of variables
    else                                           % unbound or NON-VAR
        V}                                         % return pointer
bind(U,V) term *U,V
    {if U << ST' then                              % U in the shared section
        *(++TR)=U                                  % trail U
    *U = V}                                        % bind U to V
```

Scheduling Operations

```
    create_choice_point (NumAlt) integer NumAlt
        {allocate a choice point and let B point
          to it;
        <P',ST',TR',OA'> = <P,ST,TR,NumAlt-1>;     % save state
        assign the address of the first
          alternative to P}
    backtrack
        {if not (B == bottom) then                 % more choice points
            {untrail_vars;
            <ST,TR> = <ST',TR'>;                   % restore state
            assign the address of the next
              alternative to P' and P;
            OA' = OA'-1;
            if OA' == 0 then                       % last alternative
                {remove the last choice point;     % contraction
                let B point to the previous
                  choice point}}
        else                                       % no more choice points
            terminate the computation}
```

When a choice point becomes closed, it is removed from the control stack and the private part of the (environment and term) stack is extended to the level indicated by the next choice point. We will call this operation *contraction*. The unbound variables in the extended private section of the stack can again be bound unconditionally.

```
    untrail_vars
        {while TR >> TR' do                        % below the latest choice point
            {x0=*(TR--);                           % untrail a variable
            *(x0) = tagged(VAR,x0)}}               % unbind a variable
```

In the above procedure variables are removed from the trail and their values turned back to unbound.

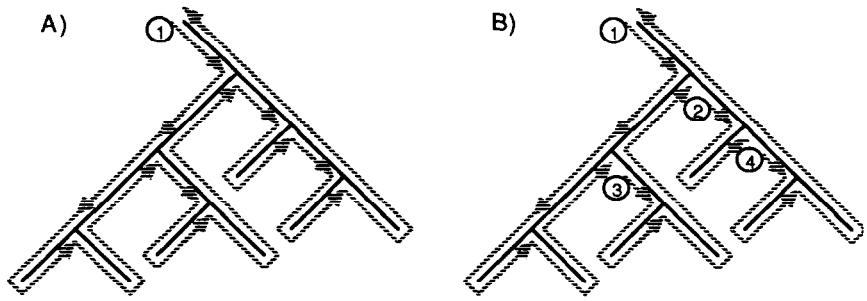


FIGURE 1. A search tree traversed by one (*A*) and four processors (*B*). Each of the processors in (*B*) will search its subtree in the depth-first manner.

3. SCHEDULING

In sequential inplementations of PROLOG the search tree of a program is traversed in a depth-first manner [Figure 1(a)]. As only one path at a time is explored, all the storage areas can be organized as stacks. In a multiprocessor implementation there are several processors performing the search simultaneously. The scheduling could be any combination of the depth-first and the breadth-first strategies. One way to divide work is to approximate the sequential processing whenever feasible. We propose that the processor which gets a subtree for processing will explore it in the depth-first manner as long as there are open nodes (choice points) in it [Figure 1(b)].

To make the scheduling complete we need some strategy to apply when the subtree of a processor is exhausted. A preferable strategy requires minimal effort and causes minimal change in the state of the processors. The effort is the examination of nodes and installation/deinstallation of variables in versions vectors (see the following section). We propose the following heuristics. The processor which has exhausted its subtree walks up the path to the root until it finds either an open node and claims an alternative from it, or finds a closed node with open nodes under it. In the second case the processor usually has several open nodes to choose among. The processor always claims an alternative from one of the nodes closest to the root (see Figure 2). We can see at least two measures for determining the closest node (or nodes). The first one is the depth of a node, and the second is the number of entries on the trail between the root and the node. Both measures would give similar behavior, though the second is more precise.

The above strategy applies even at the start time. A starting processor claims work from the node closest to the root.

Each processor has zero or more open nodes on the path from the tip it is working on to the root of the tree. We shall call the open node closest to the root the *dispatching node* of a processor. Only the part of the tree above and including the dispatching nodes is actually shared among several processors (see Figure 3). A processor looking for work can claim alternatives only from the dispatching nodes. When the dispatching node of a processor is closed (all alternatives taken), then the first node below on the branch (if any) becomes the new dispatchaing node. The subtree below a dispatching node has only one branch and is private to the

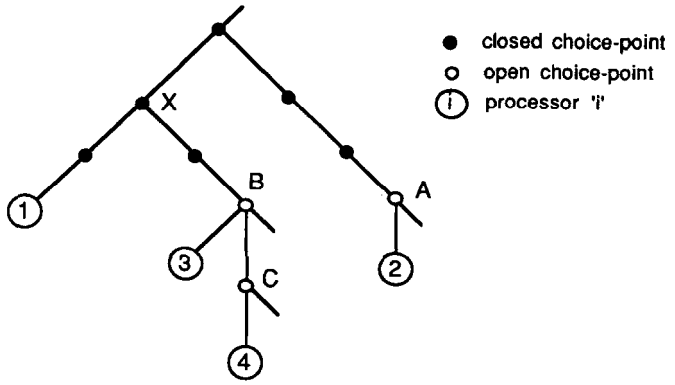


FIGURE 2. A search tree with open alternatives at nodes *A*, *B*, and *C*. When processor 1 fails, it will claim work from node *B*.

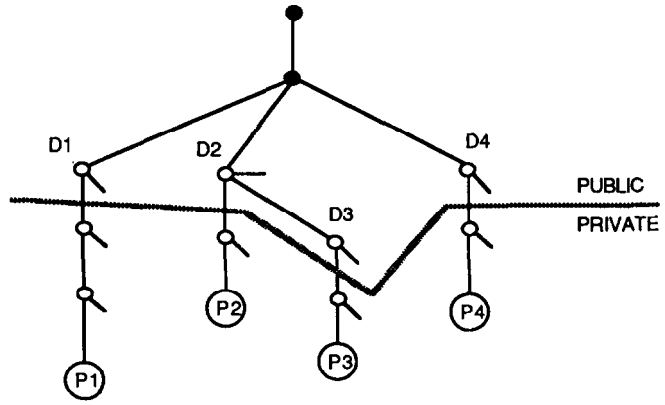


FIGURE 3. A search tree divided into the public and the private parts. Nodes *D*1, *D*2, *D*3, and *D*4 are the dispatching nodes of the corresponding processors.

processor which works on it; thus while a processor works on its private subtree, it can manage memory efficiently and has no need for locking.

In order to implement the scheduling scheme outlined above we have to extend the S-WAM. Every processor must have a pointer to its dispatching node (`DP`) and its current distance from the root (`D`). In every node there must be a record of the processors working under it (`AP'`), its distance from the root (`D'`), and the number of active paths emerging from it (`AA'`). We have given, in the parenthesis, the name of the register used for each purpose. The roles of the dispatching node and the distance have been explained above. The record of processors working under a node is used to find out which dispatching nodes should be considered. `AP'` can be implemented as a bit vector if the number of processors is moderate. The number of active paths is used to decide if the node can be removed from the tree. The use of the new components is illustrated by Figure 4.

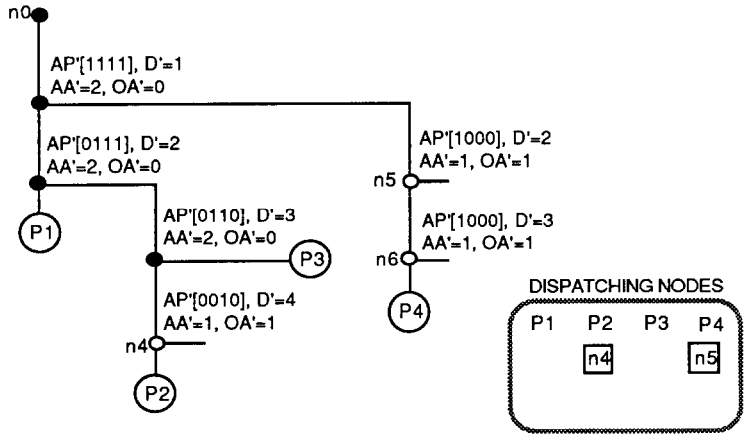
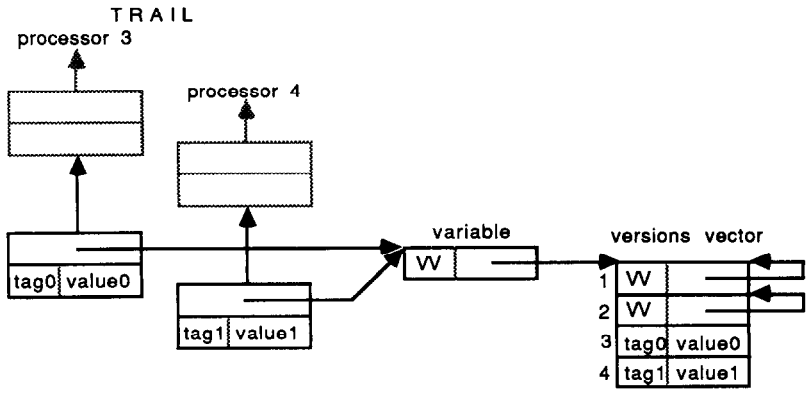


FIGURE 4. A snapshot of the information in the search tree of some program. Only part of the contents of choice points is shown. Distance is given as the depth of a node.

4. STORAGE MODEL

During sequential execution a variable can be bound conditionally to one value at a time. During OR-parallel execution a variable can have several conditional bindings simultaneously. The scheduling scheme presented in the previous section assures that the number of branches searched in parallel, and thus the number of simultaneously existing bindings to a variable, does not exceed the number of processors in the system (cases where a branch must be suspended will be discussed in Section 7). In the versions-vector model presented here we take advantage of the fact that the maximal number of active branches is fixed. The idea is to allocate a (versions) vector per conditionally bound variable. Each vector consists of a number of components equal to the number of processors in the system. Each component of a vector is used by just one processor to store and access conditional bindings belonging to that processor (see Figure 5). Access operations for all variables remain constant-time, and are only slightly more complex for conditionally bound variables.

FIGURE 5. A variable with several conditional bindings. Bindings by processors 3 and 4.



When a processor starts a new task, the conditional bindings inherited from the processor from which the task is stolen must be placed (installed) in the versions vectors' component corresponding to the starting processor. When a processor backtracks, some conditional values corresponding to that processor are removed from versions vectors (deinstalled). Deinstallation corresponds to untrailing in conventional implementations. In order to implement installation, the trail of the VV-WAM contains address-value pairs (see Figure 5) instead of just addresses. The need for installation and deinstallation is the price paid for keeping constant access time for all variables.

As mentioned earlier, D. H. D. Warren has recently proposed [17,18] the SRI model, which has properties (constant access time, nonconstant-time task switch) very similar to the VV model. Through closer scrutiny we have found out that the major difference is the allocation time for the additional memory (versions vectors in the VV model, and cells in a binding array in the SRI model) used for conditional bindings. In the SRI model a cell is allocated as soon as an unbound variable is created, and in the VV model a versions vector is allocated when a variable gets its first conditional value. In the SRI model, space is wasted for variables which will be bound unconditionally; in the VV model, components in a versions vector are allocated even for processors which will never use the variable. Intuitively the space overhead should be higher in our model, at least for a large number of processors. In addition our model requires locking on some variable accesses, while the SRI model does not. Both models will be implemented in the same system and evaluated.

5. VERSIONS-VECTOR WAM

The scheduling algorithm and the new storage model require changes to the simple WAM introduced in Section 2. In the specification below we describe mostly the differences between the VV-WAM and the S-WAM.

Data Areas

Each processor has storage areas corresponding to those of the S-WAM. The areas of different processors overlap logically, forming trees corresponding to the search tree. There is an additional area for versions vectors common for all processors. Its organization is discussed in Section 8.

Registers

Each processor has its own set of registers. There are two new registers: **DP**, pointing to the dispatching node, and **D**, keeping the distance from the root. In contrast to the old registers, **DP** and **D** in each processor can be accessed by the other processors:

P	Program pointer
ST	Top-of-stack pointer
B	Top-of-choice-point stack pointer
TR	Top-of-trail pointer
DP	Dispatching-node pointer
D	Distance counter

Contents of a Choice Point

Choice points are extended with three components (**AA'**, **AP'**, **D'**), whose functions have been explained in the preceding sections:

P'	Alternative clause
ST'	Alternative top-of-stack pointer
TR'	Alternative top-of-trail pointer
OA'	Number of alternative clauses left
AA'	Number of active paths
AP'	Active processors
D'	Distance from the root

Data Objects

As in the S-WAM, a PROLOG term is represented by a tagged word. A variable having several versions of bindings is bound to a versions vector with, possibly, different values in different components. Such a variable is represented by a word tagged with a new tag **VV**. A variable unbound for processor **I** is represented either by a reference to itself (unbound in all processors having access to it), or by a pointer to a versions vector with the component **I** bound to itself (see Figure 5).

Operations

All the operations have been modified. The scheduling operations are extended with the procedure **steal** invoked when there is no work left in the branch. We have added one important procedure, **install_task**, invoked before starting a new task in order to install bindings and to add the processor identifier to the choice points on the installation path.

Operations on a Single Variable. The **deref** procedure must consider the case of variable bound to a versions vector. In that case the address of the component is computed using the processor id (**Pid**). Notice that the pointer to a value cell on the stack is returned even if a variable is unbound so far as the accessing processor is concerned, but bound in some other processors.

```
deref(V,Pid) term *V, integer Pid
  {if tag(*V) == VAR then
       {if not value(*V) == V then
          deref(*V,Pid)
       else                                    % unbound on the stack
          V}
  else if tag(*V) == NON-VAR then              % bound
          V
  else                                         % tag(*V) == VV, variable bound
                                                 to a versions vector
       {mv = value(*V) + Pid;
       if not value(*mv) == mv then            % bound
          deref(*mv,Pid)
       else                                    % unbound
          V}}                                  % pointer to the value cell
```

A variable can be bound (by **bind**) unconditionally only if it is in the private section. Otherwise the address and the new value of the variable are trailed and the value is stored in the proper component of the corresponding versions vector. If it is the first binding to that variable, then a versions vector is allocated. Notice that the vector need not be initialized on every allocation. All the vectors are initialized prior to or at the first allocation and are cleaned up during untrailing (see **untrail_vars**):

```
bind(U,V,Pid) term *U,V, integer Pid
   {if U << ST' then                                  % in shared section
      {*(TR++) = <U,V>;                               % trail a pair
      if value(*U) == U then                          % totally unbound
         {allocate a versions vector
            and assign its address to vec;
         *U = tagged(VV,vec)}                         % bind U to the
                                                        versions vector
      else
         vec = value(*U);
      vec(Pid) = V}                                   % bind a component
   else                                               % in the private section
      *U = V}
```

The operations on variables have become more complex, but the overhead is invoked *only* for variables being bound conditionally (**bind**) or already bound (**deref**) in versions vectors.

Scheduling Operations. Under the subtitle "scheduling operations" we specify how the control tree is expanded and contracted, how it is searched, how the dispatching registers are changed, and finally how installations and deinstallations are done.

The **create_choice_point** procedure initializes the components of a choice point. If the processor has no dispatching node, this choice point becomes one. The calling processor becomes the first active one under that choice point:

```
create_choice_point(Pid,NumAlt) integer Pid,NumAlt
   {allocate a choice point and let B point to it;
   if DP == null then                                 % no dispatching node
      DP = B;
   <P',ST',TR',OA',D'> = <P,ST,TR,NumAlt-1,D>;
   AA' = 1;
   D = D+1;                                           % increase the
                                                        depth counter
   AP'(Pid) = true;                                   % the first
                                                        active processor
   assign the address of the first alternative to P}
```

The **backtrack** procedure (Table 1) has two main cases: the first (**OA'** > 0) specifies the proper backtracking, and the second (**OA'** = 0) specifies the search for work after the private subtree is exhausted.

In the first main case there are alternatives to be taken from the choice point. The basic actions are the same as during sequential backtracking (untrail variables, restore state, and possibly remove the choice point). There are two important subcases. In the first (**AA'** = 0) there are no more active branches under that choice point, and thus no more processors. In the second there are more active branches below. Only in the first case can the backtracking processor free some of the

TABLE 1

```
backtrack(Pid,Alone) integer Pid, bool Alone
    {if Alone then
        AA' = AA'-1;
    if OA' > 0 then                          % open choice point
        {OA' = OA'-1;
        if AA' == 0 then                     % the only processor under
            {if OA' == 0 then                  this choice point
                untrail_vars(Pid,true)       % untrail and clean up
            else
                untrail_vars(Pid,false);     % just untrail
            <ST,TR> = <ST',TR'>;
            assign the address of the next
                alternative to P and P';
            if OA' == 0 then                 % the last open alternative
                {if DP == B then             % dispatchaing node closed
                    DP = null;
                remove the last choice point;
                let B point to the previous
                  choice point}
                AA  = 1}
        else                                 % not the only processor
            {untrail_vars(Pid,false);        % just untrail
            <ST,TR,D> = <ST',TR',D'>;
            assign the address of the next
                alternative to P and P';
            if OA' == 0 and DP == B then     % dispatching node closed
                DP = null;
            AA' = AA'+1}
    else                                     % closed choice point
        if AA' == 0 then                     % no subtrees below
            {untrail_vars(Pid,true);         % untrail and clean up
            remove the last choice point;
            let B point to the previous
              choice point;
            backtrack(Pid,true)}             % go on upwards, still alone
        else                                 % a subtree below
            {untrail_vars(Pid,false);        % just untrail
            steal(Pid);                      % look for work, return only if
                                               nothing to steal
            AP'(Pid) = false;                % no longer active below
            let B point to the previous
              choice point;
            backtrack(Pid,false)}}}          % go upwards, no longer alone
```

versions vectors and deallocate the choice point if it is taking the last alternative. It is very important to remove choice points whenever possible, because then the private section of the (binding) stack can be extended, and the need for versions vectors decreases. Versions vectors are removed when the tree is contracted. They could be removed on each untrailing (the processor is alone in this section anyhow), but that would cause frequent occurrences of deallocation followed shortly by allocation for the same variable. Notice that the **DP** of the processor is updated by the owner when the dispatching node is closed and removed.

TABLE 2

```
steal(Pid) integer Pid
    {pid' = find an open dispatchaing node DP(pid'),
    where
       pid' in AP', with the minimal depth D'(pid');
    if pid' == none then                                    % not found
       {if B == root then
          {if all other processors waiting then
             terminate computation
          else wait until a parallel choice point
            is created}
       else
          return}                                           % to continue
                                                              backtracking
    crosschp = B;
    B = get the address of the chosen dispatching node;
    <ST,TR,D> = <ST',TR',D'>;
    DP = null;
    assign the address of the next alternative to P and P';
    OA' = OA'-1;
    if OA' == 0 then
       inform the owner that its dispatchaing
          node has been closed;
    AA' = AA'+1;                                            % new active path
    install_task(B,crosschp,Pid)}                           % add bindings and Pid
```

In the second main case the processor contracts a branch as long as it is alone on it (**AA'** = 0). In the course of contraction variables are untrailed, and versions vectors and choice points decallocated. When a node with active processors under it is reached, the processor tries to steal an alternative. If it fails, it continues climbing up the tree after removing itself from the record of active processors. Notice that the active-path counter (**AA'**) and the argument **Alone** are used together as a distributed counter of active branches.

The **steal** procedure (Table 2) is responsible for finding an open alternative and, if found, setting up a new task. In cases where there are no open alternatives and no other active processors, the computation terminates. If an open choice point is found, the new task is set up by loading the registers of the calling processor from the choice point, adjusting the contents of the choice point, installing variables on the trail between the open choice point and the choice point where the search stopped (cross choice point), and finally adding the processor id to all choice points on the way. When the last alternative is stolen, the owner of the node must be informed that its dispatching-node pointer must be updated. Letting the stealing processor do the updating would require an intricate locking protocol.

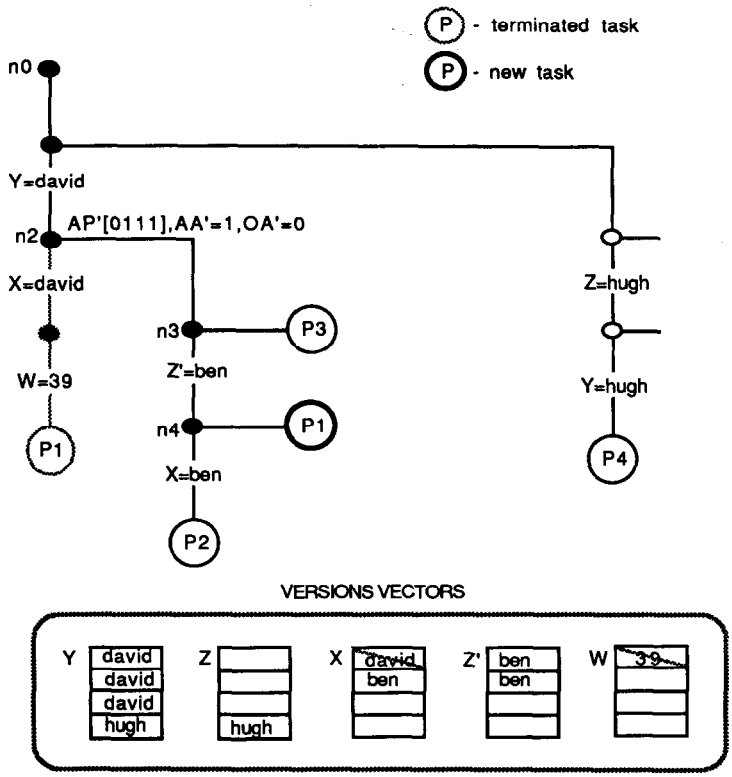
The **untrail_vars** procedure removes variables from the trail, makes the proper component in the versions vectors unbound, and lastly removes versions vectors for variables in the section of the stack which has become private or will be deallocated. A section becomes private when the last open alternative is taken and the choice point is removed by the processor taking the alternative (case **OA'** = 1, **AA'** = 0 in **backtrack**). A section is deallocated when the last active processor in a subtree passes a closed choice point looking for work (case **OA'** = 0, **AA'** = 0 in

backtrack):

```
untrail_vars(Pid,Last) integer Pid, bool Last
    {if Last then
        {assign the address of the previous
        choice point to chp;
        new_limit = chp->ST'}
    while TR >> TR' do
        {<x0,_> = *(TR'--);                    % get address from trail
        vec = value(*(x0));
        vec(Pid) = tagged(VAR,vec+Pid);        % unbind a component
        if Last and x0 >> new_limit then       % cleaning up
            deallocate the versions vector
            pointed by vec;                    % notice that all
        }}                                       components are unbound
```

The `install_task` procedure copies values from the part of the trail, which is between the choice point from which the alternative is stolen and the cross choice point, to the proper component (`Pid`) of concerned versions vectors. In addition, information about the new active processor is added to all the choice points between

FIGURE 6. Backtracking, stealing, and installation. Variables on the trails are shown on appropriate arcs. Processor $P1$ has terminated a task and found a new one. Variables X and W have been untrailed, and Z' has been installed.



the ones mentioned:

```
install_task(Bsteal,Bcross,Pid) pointer Bsteal,Bcross,integer Pid
    {b1 = Bsteal;
    t1 = b1->TR';
    loop
        {b2 = address to the choice point preceding b1;
        t2 = b2 ->TR';
        while t1>> t2 do
            {<x0,value0> = *(t1--);
            vec = value(*(x0));
            vec(Pid) = value0}                          % bind a component
        b1->AP'(Pid) = true;                            % add processor
        exit if b2 == Bcross;
        b1 = b2;
        t1 = t2}}
```

We conclude this fairly technical section by an example illustrating backtracking, stealing and installation (see Figure 6).

5. OPTIMIZATIONS

The cost of task switching is a major overhead in the VV-WAM. We propose two optimizations decreasing that cost. The first, *straightening and promotion*, consists of removing nodes from the control tree (straightening) and making some variable unconditionally bound (promotion). It is done during the installation phase of a task switch and saves time for the subsequent processors starting tasks in the same subtree. It also decreases the search time. The second, *delayed installation*, postpones installation until variables are actually accessed. This mainly decreases the cost of starting short branches.

5.1. Straightening and Promotion

In the sequential WAM a node which is closed can be immediately deallocated. In the VV-WAM there are three cases. In the first case a tip node is closed and deallocated; that case corresponds to sequential execution and has been dealt with in the previous section (contraction). In the second case an internal node with more than one arc below is closed, but must not be removed from the tree because it is still needed for scheduling. In the third case, dealt with in this section, an internal node with just one arc below is closed and can be removed. We will call such node *dead*, and say that the branch is straightened when such a node is removed. Removing dead nodes decreases the task switching time because then fewer nodes must be searched. Moreover, when a branch is straightened, bindings of the variables on the arc between the dead node and the node preceding it (promoted arc) and in the trail segment between the dead node and the one following it can be promoted. During promotion, bindings are copied from the trail to the value cells, the corresponding variables untrailed, and the corresponding versions vectors deallocated. Moving bindings decreases access time to variables somewhat, untrailing decreases installation and deinstallation time considerably, and deallocation saves space. Our main aim is to save installation and deinstallation time; thus it is sufficient to start promotion during installation of the first task taken from a choice point below the dead one. Subsequent processors fetching alternatives in this

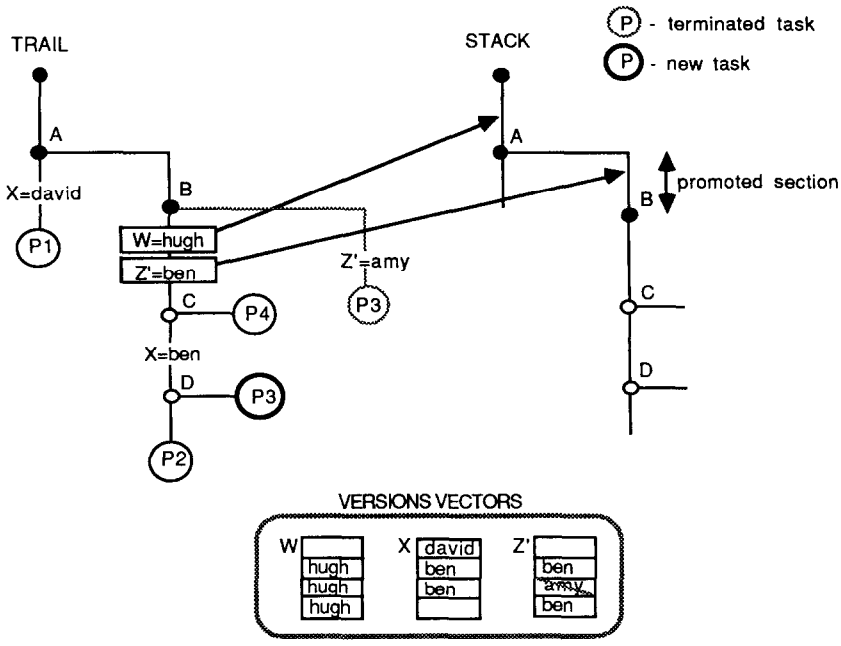


FIGURE 7. Processor 3 has terminated a task and passed the choice point B on backtracking. B became dead. When an alternative is stolen from the choice point D, the binding of the variable Z' in the promoted section is made unconditional. Arrows point to the part of the stack where value cells for the variables are located.

subtree will have less variables to install and deinstall. Promotion is illustrated by Figure 7.

Effects of straightening are limited to nodes. Promotion is more critical because it is concerned with variables. In this paper we shall only discuss implementation of promotion.

The process of promotion is complicated by the fact that the promoted bindings, the corresponding trail entries, and the versions vectors are usually shared among several processors. We have designed a promotion algorithm so that no locking is needed in spite of sharing. The idea is that no locking is required as long as the promoted bindings do not disappear from versions vectors until it is sure that the concerned processors cannot access them, i.e., until backtracking.

To achieve this we replace the trail by two separate trails: a trail for installation (installation trail), and a trail for cleaning up and removing versions vectors (cleanup trail). An element in the installation trail is a pair ⟨value cell address, value⟩, and an element in the cleanup trail is a pair ⟨versions-vector address, record of processors with bindings in this vector⟩. The record can be implemented as a bit map.

When a variable is bound conditionally, an entry is added to both trails. When a binding of a variable is copied to its value cell, the corresponding entry is removed from the installation trail, but its cleanup-trail entry and its versions vector are not

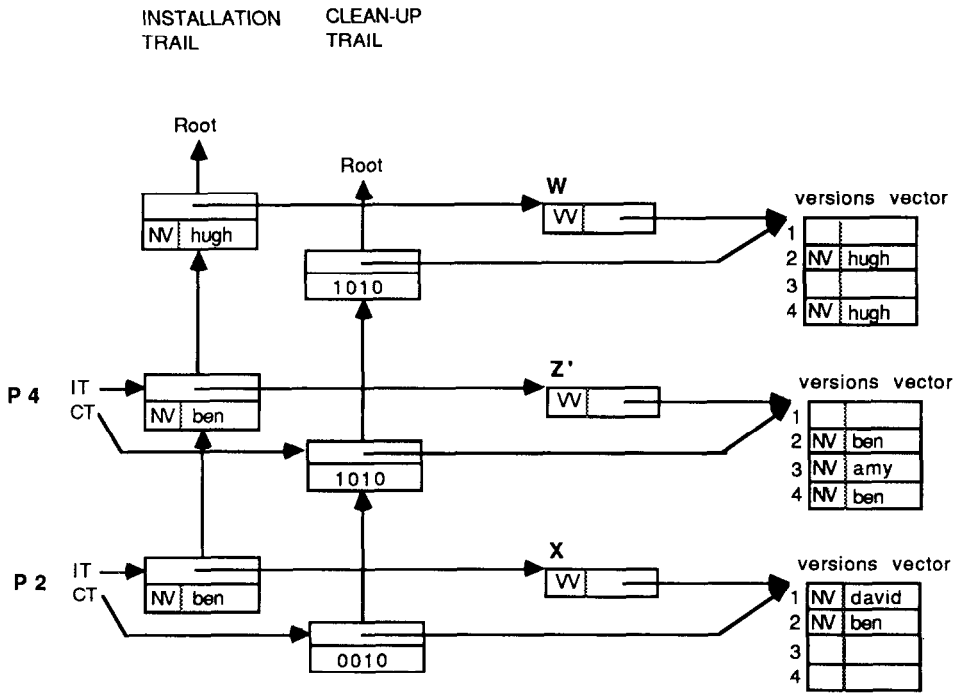


FIGURE 8A. Before promotion. The trails of processors $P2$ and $P4$ are shown (see Figure 7). Processor $P3$ has not started backtracking yet. IT and CT are pointers to installation and cleanup trails. NV means **NON-VAR**.

changed. The cleaning up and removing of versions vectors is delayed until backtracking, when the cleanup trail is used. Because of the delay, processors that entered the subtree after promotion will have irrelevant entries in the cleanup trail. It is important to avoid versions-vector operations for such processors. That is done by recording on the cleanup trail the identities of the processors adding bindings to versions vectors. A processor's identification is added to a record when a variable gets conditionally bound or the binding is installed, and is removed when the processor backtracks. Backtracking processors clean the versions vectors' entries, and finally deallocate the versions vectors and the trail entries, but only for the trail entries containing their identities.

The process of promotion is illustrated in Figure 8(a) and (b) (the data structures shown correspond to the situation in Figure 7).

The promotion algorithm can be further optimized by grouping the entries in the cleanup trail between two nodes into the promoted entries and the nonpromoted ones. One membership test per each group of promoted entries, will then be enough, and no other test will be required. The ideas of associating a record of processors with each trail entry and of grouping the entries has been borrowed from [11].

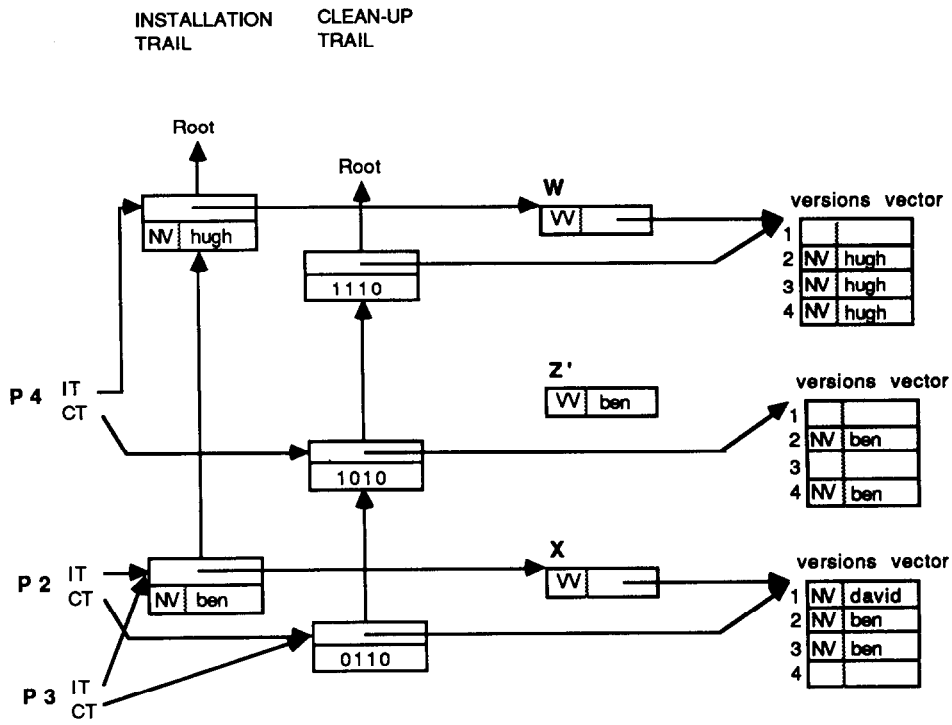


FIGURE 8B. After promotion. Processor 3 has backtracked, cleaning its entry in Z"s versions vector. Thereafter $P3$ entered the subtree rooted in node B (see Figure 7), promoted Z', and installed the bindings for W and X. Note that if $P3$ fails, then while backtracking it will not have to inspect the versions vector for Z' ($P3$ is not in the record of processors with bindings in this vector). If later $P4$ and $P2$ backtrack, they will clean and finally remove Z"s trail entry and versions vector.

5.2. Delayed Installation

The overhead in starting short branches is troublesome in any implementation. The problem becomes even more serious in our implementation because of the expense of setting up a new task. One means of avoiding short branches is indexing. Unfortunately, even with indexing (especially on the first argument only) there will still exist branches that will fail shortly after creation. We propose delayed installation, mainly to decrease the cost of such branches.

The delayed installation means that the bindings on the trail are not moved to the versions vector when a task is set up, but only when a noninstalled conditionally bound variable is accessed. The procedure will install all the variables on the relevant part of the trail up to the accessed variable. There is no overhead on other accesses. Figure 9 shows a case of delayed installation.

6. RESTRICTING PARALLELISM

We have assumed, so far, that all branches in a search tree are always explored and always (potentially) in parallel. It must be possible to prohibit or stop execution of

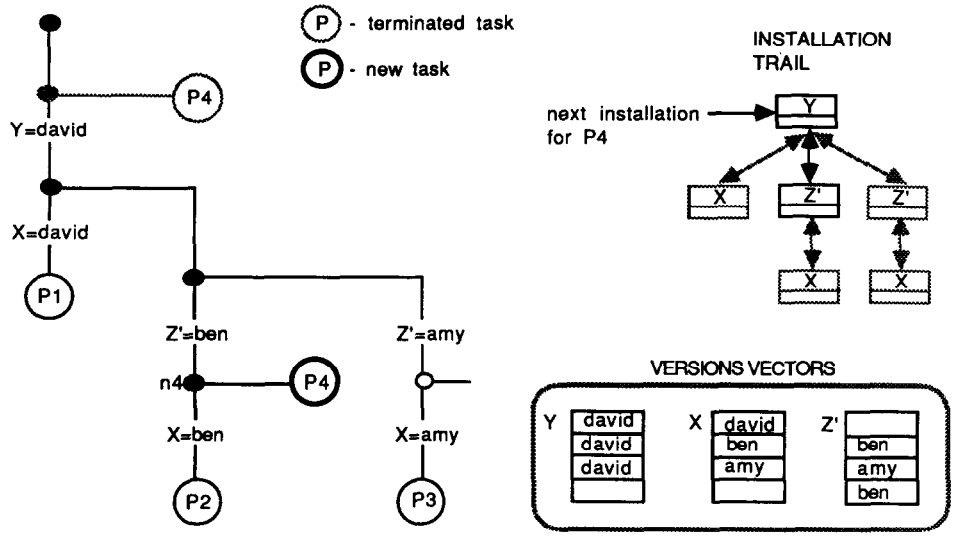


FIGURE 9. Processor *P4* has terminated a task and started a new one from the choice-point *n4*. Installation of the variables *Y* and *Z'* has been delayed. The figure shows a snapshot when *Z'* has already been accessed and installed. The pointer to the trail shows where the installation will proceed later on.

some branches. That can be done using constructions like cut or commit. There are also several reasons (e.g. efficiency, algorithms, side effects) for allowing a programmer to annotate programs so that only clauses in some predicates will be allowed to be executed in parallel.

Implementation of cut/commit in a parallel environment is quite intricate and will be described elsewhere. It is enough to say here that the solution requires suspension of branches when cut/commit are considered together with global side effects (assert, I/O). That violates our assumption about the number of branches explored simultaneously not exceeding the number of processors.

That problem can be solved in several ways. One solution is to let the processor suspending a branch idle until the branch is activated again. It is simple, but could severely limit the number of available processors. Another solution is to limit the number of simultaneous branches to some arbitrary number larger than the number of processors. The usefulness of that solution depends upon the typical number of simultaneous suspensions. There is, finally, a general solution. When a branch is suspended, the processor working on it can clean up the *relevant* versions vectors and look for some other work (backtracking without cleaning up the trails). The suspended branch can be executed later on by some other processor, using the values left on the installation trail to set up the new task. If suspension is not a very frequent operation, this solution is preferable.

Parallel and sequential annotations can be used to increase the efficiency of the VV-WAM. The existence of two types of predicates implies parallel and sequential choice points. Alternatives can only be stolen from parallel choice points. We utilize that fact to decrease the number of variables to be installed on a task switch (even dereferencing chains will be shortened, but it is only a marginal gain). Observe that

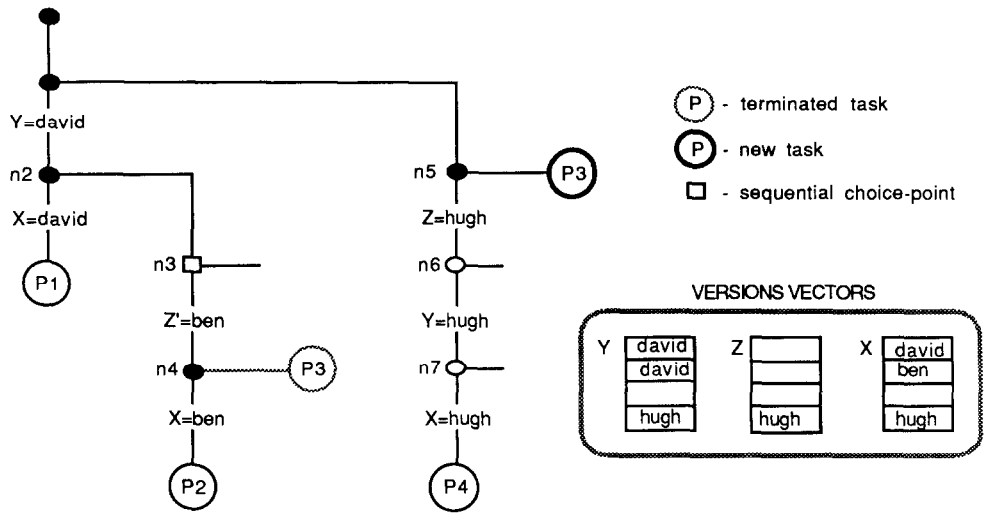


FIGURE 10. Sequential and parallel choice points. $N3$ is a sequential choice point; all the others are parallel. Processor $P3$ has finished one task. It may not take the alternative from $n3$, because $P2$ is still working below. Instead it goes on to $n2$ to finally find an open alternative in $n5$. The open alternative in $n3$ will be executed by $P2$. The variable Z' was bound before the parallel choice point was created, and thus does not have a versions vector.

only the variables which are unbound when a parallel choice point is created may get several simultaneous bindings. If there are one or more sequential choice points below the last parallel choice point, variables in the stack section below the parallel one might take on several values, but not simultaneously (just like variables in the shared section of a sequential implementation), and thus do not require versions vectors and do not have to be installed (their values are stored directly in value cells).

The existence of two types of choice points complicates the scheduling slightly. A processor cannot take an alternative from a sequential choice point if there are active branches below it, because it would unbind variables on the shared section of the stack. The processor will instead go to the closest parallel choice point and go on with the backtracking procedure. The sequential alternatives will be taken by the last processor in that subtree.

Figure 10 shows the situation with a sequential choice point temporarily "trapped" above a parallel choice point.

7. LOW-LEVEL CONSIDERATIONS

Any specification stops at some level, hiding (more or less) details from the reader. Our specification hides two important issues: synchronization and storage management. We shall discuss them briefly below.

Synchronization is an issue not present in sequential implementations of PROLOG. Synchronization not only takes time in the form of instructions executed, but can also cripple parallelism. We have designed our algorithms taking great care

to avoid synchronization. In the VV-WAM no synchronization is needed in the management of variables, with one exception. When a variable in the shared section is about to be conditionally bound and is not yet bound to a versions vector, the variable cell must be locked until the vector is allocated and the cell bound to it. As mentioned earlier, the extra trail in the implementation of promotion has been introduced in order to avoid locking in promotion. An alternative would be to deallocate versions vectors immediately when the corresponding variables are promoted, but that would make locking necessary on *all* read accesses (dereferencing) to the shared section, which we found unacceptable.

We have nearly managed to avoid locking in operations on variables, but complex synchronization is still needed in scheduling operation, not least when cut/commit and side effects are considered, as described in [10].

All areas, except for the versions-vector area, are managed similarly to the corresponding areas in sequential implementations, i.e. as stacks. As mentioned earlier, each processor has an instance of each area. A processor always allocates space in one of its own areas, though it can write and read in other processors' areas. The fact that the areas are logically shared complicates storage deallocation, because it may happen that there are objects in a processor's area which are no longer needed by this processor but still used by other processors.

The versions-vector area cannot be managed as stack, because each versions vector contains components for all processors. A simple physical organization is to have a list of free versions vectors for each processor. A processor allocating or deallocating a vector would always use its own list. The lists would have to be rebalanced when some processor runs out of free vectors.

We are also considering a compromise implementation using binding arrays. In this implementation, space for conditional bindings is allocated first when a section of the stack becomes shared (a choice point is created), that is, later than in the SRI model but earlier than in the current VV model (see Section 4). The space is allocated only for variables which are unbound when they become shared, and only for processors actually using these variables. Such implementation would have smaller memory consumption than any of the other models, and only slightly more complex operations on variables.

Another property of PROLOG implementations is the need for seniority tests among objects. In sequential implementations the age of an object is decided by its address. With the storage organized as described above, the address of an object no longer reflects its age. A simple and efficient solution is to store the age of an object in the object itself. For the choice point the depth `D'` gives its age. The depth can also be used to decide the age of variables. The depth is stored in the value cell of a variable as long as the variable is totally unbound, and later stored in the corresponding versions vector.

8. CONCLUSIONS

Our goal is to execute a superset of PROLOG on shared-memory multiprocessors so that no programs run much slower and most programs run much faster than on single processor system. The abstract machine we propose brings us closer to that goal. It can execute sequential programs with nearly no overhead. It allows simulta-

neous execution of several branches with very little overhead in operations on variables. It actually assures that operations on all variables are constant-time operations. The cost of starting a new task depends on the distance in a search tree to be covered during installation of bindings. By the use of promotion, delayed installation, and sequentialization we bring that cost down. The promotion prohibits the cost from growing when the tree grows, the delayed installation makes execution of short branches cheap, and sequentialization decreases the number of variables requiring versions vectors. The main disadvantage of our proposal is the (possibly) high memory consumption. The memory consumption depends on the fraction of variables requiring versions vectors. All optimizations described in the preceding sections bring this fraction down.

Many of the issues presented in this paper are common to all parallel implementations of PROLOG related languages. Many of the presented solutions can be applied when implementing other models.

Our effort has been part of an informal cooperation among Argonne National Laboratory, University of Manchester and SICS. The cooperation has resulted in several different storage model and scheduling algorithms. Some have already been implemented, giving very encouraging results [9].

This work would not be possible without the exciting cooperation, called the Gigalips project, among Argonne National Laboratory, University of Manchester, and Swedish Institute of Computer Science. We are especially grateful to Ross Overbeek, David H. D. Warren, and Mats Carlsson for their constructive criticism.

REFERENCES

1. Borgwart, P., Parallel Prolog Using Stack Segments on Shared-Memory Multiprocessors, in: *Proceedings of 1984 International Symposium on Logic Programming*, Atlantic City, pp. 2–11.
2. Carlsson, M., Internals of SICStus Prolog, Draft, Nov. 1987.
3. Ciepielewski, A. and Haridi, S., A Formal Model for OR-Parallel Execution of Logic Programs, in *IFIP 83*, (P. C. Mason, Ed.), North Holland, pp. 299–305.
4. Ciepielewski, A. and Haridi, S., Storage Models for OR-Parallel Execution of Logic Programs, CSLAB Working Paper 821121, TRITA-CS-8301, Royal Inst. of Technology, Stockholm, 1983.
5. Disz, T., Lusk, E., Warren, and Overbeek, R., Experiments with OR-Parallel Logic Programs, in: *Proceedings of 4th International Conference on Logic Programming*, Melbourne, 1987, pp. 576–600.
6. Gregory, S., Design, Application and Implementation of a Parallel Logic Programming Language, Ph.D. Thesis, Dept. of Computing, Imperial College, Univ. of London, Sept. 1985.
7. Hermenegildo, M. V., An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel, Ph.D. Thesis, Univ. of Texas at Austin, Aug. 1986.
8. Lindstrom, G., OR-Parallelism on Applicative Architectures, in: *Proceedings of the Second International Logic Programming Conference*, Uppsala, 1984, pp. 159–170.
9. Lusk, E., Warren, D. H. D., Haridi, S., Butler, R., Calderwood, A., Disz, T., Olson, R., Overbeek, R., Stevens, R., Szeredi, P., Brand, P., Carlsson, M., Ciepielewski, A., and Hausman, B., The Aurora OR-parallel PROLOG System, in: *Proceedings of International Conference on Fifth Generation Computer Systems 1988*, pp. 819–830, ICOT, Nov. 1988.

10. Overbeek, R., ANL-WAM Synchronization Issues, personal communication, Feb. 1987.
11. Overbeek, R., Implementing Promotion for the Gigawam, personal communication, Nov. 1987.
12. Pollard, G. H., Parallel Execution of Horn Clause Programs, Ph.D. Thesis, Imperial College of Science and Technology, Univ. of London, 1981.
13. Shapiro, E., Concurrent Prolog: A Progress Report, *Computer*, Aug. 1986, pp. 44–48.
14. Tinker, P. and Lindstrom, G., A Performance-Oriented Design for OR-Parallel Logic Programming, in *Proceedings of 4th International Conference on Logic Programming*, Melbourne, 1987, pp. 601–615.
15. Ueda, K., Guarded Horn Clauses, Ph.D. Thesis, Information Engineering Course, Faculty of Engineering, Univ. of Tokyo, 1986.
16. Warren, D. S., Efficient Prolog Memory Management for Flexible Control Strategies, in: *Proceedings of 1984 International Symposium on Logic Programming*, Atlantic City, pp. 198–202; also in *New Generation Comput.*, No. 4, 1984.
17. Warren, D. H. D., OR-Parallel Execution Models of Prolog, in: *Proceedings of Joint Conference on Theory and Practice of Software Development*, Pisa, 1987.
18. Warren, D. H. D., The SRI Model for OR-Parallel Execution of Prolog—Abstract Design and Implementation Issues, in: *Proceedings of 1987 Symposium on Logic Programming*, San Francisco, pp. 92–102.
19. Warren, D. H. D., An Abstract Prolog Instruction Set, SRI Technical Note 309, Oct. 1983.
20. Warren, D. H. D., Prolog Implementation and Architecture, Tutorial notes from the 3rd International Logic Programming Conference, London, 1986.